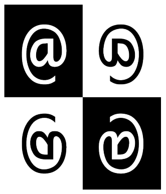
15-122: Principles of Imperative Computation, Fall 2015

Programming 3: Images



Due: Thursday, September 24, 2015 by 22:00

This programming assignment will have you using arrays to represent and manipulate images.

The code handout for this assignment is at

`http://www.cs.cmu.edu/~fp/courses/15122-f15/assignments/images-handout.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a FIVE (5) PENALTY-FREE HANDIN LIMIT, with the idea that for each task you can test your code, hand in, and then fix any bugs found by autolab while working on and testing the next task. Make sure to leave enough submissions to work on the optional Task 5, if you wish to do that. Every additional handin will incur a very small (0.2 point) penalty.

Style Grading: With this assignment, we will begin to emphasize *programming style* more heavily. We will actually be looking at your code and evaluating it based on the criteria outlined at `http://www.cs.cmu.edu/~rjsimmon/15122-s15/etc/styleguide.pdf`. We will make comments on your code via Autolab, and will assign an overall passing or failing style grade. A failing style grade will be temporarily represented as a score of -15. This -15 will be reset to 0 once you:

1. fix the style issues,
2. see **any** member of the course staff during office hours, and
3. briefly discuss the style issues and how they were addressed.

We will evaluate your code for style in two ways. We will use `cc0` with the `-w` flag that gives style warnings – code that raises warnings with this flag is almost certain to fail style grading. Because the `-w` flag does not check for good variable names, appropriate comments, or appropriate use of the functions defined in `pixel.c0` and `imageutil.c0`, these issues will be checked by hand.

Task 1 (2 pts) *In addition to using good style, be sure to include appropriate contracts, `//@requires`, `//@ensures`, and `//@loop_invariant`. Your annotations should at least be sufficient to ensure that all your array accesses are safe, and part of your grade will be based on a visual inspection of this.*

1 Image manipulation

The two programming problems you have for this assignment deal with manipulating images. An image will be stored in a one-dimensional array of pixels. (The C0 image library assumes the ARGB implementation of pixels that you wrote last week.) Pixels are stored in the array row by row, left to right starting at the top left of the image. For example, if a 5×5 image has the following pixel "values":

$$\begin{matrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{matrix}$$

then these values would be stored in the array in this order:

$$a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p\ q\ r\ s\ t\ u\ v\ w\ x\ y$$

In the 5×5 image, the pixel i is in row 1, column 3 (rows and columns are indexed starting with 0) but is stored in the one-dimensional array at index 8. An image must have at least one pixel.

Task 2 (2 pts) *Complete the C0 file* `imageutil.c0`*. As with the* `pixel.c0` *implementation from last week, you must fill in the missing code and translate the English preconditions and postconditions into* `//@requires` *and* `//@ensures` *statements.*

We do not require you to hand in a `images-test.c0` file that tests your imageutil implementation the way you tested your pixel implementation. It would be a good idea to write one to test your own implementation, however!

2 Image Transformations

The rest of this assignment involves implementing the core part of a series of image transformations. Each function you write will take an array representation of the input image and return an array representation of the output image. These functions should *not* be destructive: you should make your changes in a copy of the array, and not make any changes to the original array. Your implementations should be relatively efficient, meaning both that they should have a reasonable big-O running time and that they should take at most a few seconds to run on our example images.

Remember that your code should have appropriate preconditions and postconditions. It is always a precondition that the given width and height are a valid image size that matches the length of the pixel array passed to the function. It is always a postcondition that the returned array is a different array that the one that was passed in, and that this resulting array has the correct length.

In order to pass style grading, you will be expected to use functions from the pixel interface (the type `pixel` and functions `get_red`, `get_green`, `get_blue`, `get_alpha`, and `make_pixel`) and the imageutil interface (the functions `is_valid_imagesize`, `get_row`,

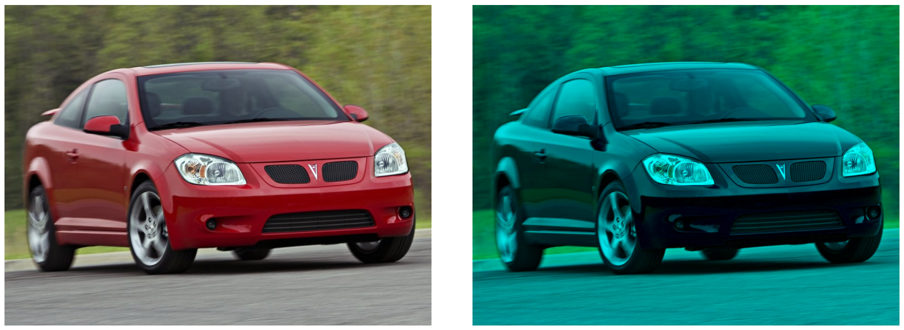


Figure 1: A sporty coupe before and after red removal.

get_column, is_valid_pixel, get_index) in the next two tasks. On Autolab, we will compile your code for tasks 2 and 3 against *our* implementation of the pixel and the imageutil interfaces, so you cannot add new functions to these interfaces.

Testing. You should use the provided `*-main.c0` files to help you test your code. The use of these files is described in the `README.txt` in the code handout.

For this assignment, we are providing a program, `imagediff`, to help you compare your output images to the sample images in the handout, optionally saving an image that shows you exactly where the two images differ. It is in the course directory on `afs`, so it is available on any cluster machine or when you are connected via ssh. For example:

```
imagediff -i images/sample.png -j images/my-image.png -o images/diff.png
```

This command compares the image `images/sample.png` and `images/my-image.png` and creates a visual representation of the difference in `images/diff.png`.

2.1 Removing red

As an example of image manipulation, you should take a look at `remove-red.c0`. The core of this transformation is this function:

```
    pixel[] remove_red (pixel[] pixels, int width, int height)
```

An example of this transformation is given in Figure 1.

You should look at `remove-red.c0` to get an idea of how this transformation works, and you should look at `README.txt` to see how to compile and run this transformation against `remove-red-main.c0`. You are strongly encouraged to write some smaller test cases for your programs. An example of what this should look like is given in `remove-red-test.c0`.

Note that `remove-red.c0` doesn't use the pixel or imageutil libraries. If *your* code doesn't use the pixel or imageutil libraries, you will fail style grading! While it is not required, you might want to try your hand at modifying `remove-red.c0` to use the pixel and imageutil libraries.

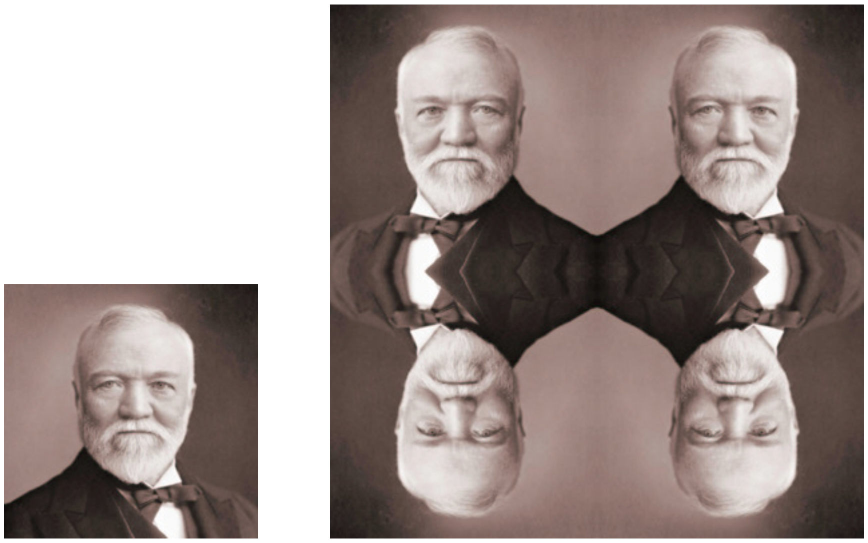


Figure 2: Original image (left); Image after "reflection effect"

2.2 Reflection Effect

In this problem, you will create a reflection effect on an image. The core of this transformation is this function:

```
pixel[] reflect(pixel[] pixels, int width, int height);
```

Your task here is to implement a function that takes as input an image of size $w \times h$ and creates an output image four times as large. The top right quadrant of the output image will contain the original image, the top left will contain the input image reflected across the y-axis, the bottom right will contain the input image reflected across the x-axis, and the bottom left will contain the input image reflected across both axes. A sample image is shown in Figure 2.

If the original image has size $w \times h$, the returned image should have size $2w \times 2h$. If the supplied array does not exactly match the size given by the width and height, your function should abort with a precondition failure when compiled and run with the `-d` flag.

Task 3 (5 pts) *Create a C0 file* `reflect.c0` *implementing the function* `reflect`*. You may include any auxiliary functions you need in the same file, but you should not include a* `main()` *function.*

You should look at `README.txt` to see how to compile and run this transformation using `reflect-main.c0`. You are also strongly encouraged to write some test cases for your programs in `images-test.c0`.

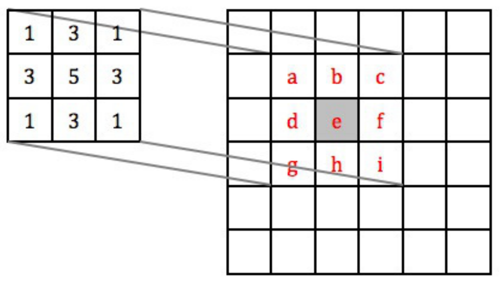


Figure 3: Overlay the 3 X 3 mask over the image so it is centered on pixel e to compute the output value corresponding to pixel e.

2.3 Blurring an Image

In this task, you will write a function to blur an image. The core of this transformation is this function:

```
pixel[] blur(pixel[] pixels, int width, int height,
             int[] mask, int maskwidth)
```

The returned array should be the representation of the blurred image.

Masks In addition to an input image, we pass the blur transformation a *mask*, an $n \times n$ array of non-negative integers representing *weights*. For our purposes, n must be odd. This means that the $n \times n$ array has a well defined center – the *origin*. While weights in the mask can be 0 (but not negative), the weight in the center position of the mask cannot be zero.

For each pixel in the input image, think of the mask as being placed on top of the image so its origin is on the pixel we wish to alter. The original intensity value of each pixel under the mask is multiplied by the corresponding value in the mask that covers it. These products are added together, and then we divide by the total of the weights in the mask to get the intensity of the output pixel. Always use the original (input) values for each pixel for each mask calculation, not the output values you compute as you create the new image.

For example, refer to Figure 3, which shows a 3×3 mask and an image that we want to blur. Suppose we want to compute the output intensity value based on pixel `e`. Imagine overlaying the mask so its center position is on `e`. We would compute the output intensity for e as:

```
(a + 3b + c + 3d + 5e + 3f + g + 3h + i)  / 21
```

This calculation is done three times for each pixel, once for its red channel, once for its green channel, and once for its blue channel. Do not apply the transformation to the alpha channel of the pixel.

Note that sometimes when you center the mask over a pixel you want to blur, the mask will hang over the edge of the image. In this case, compute the weighted sum of only those pixels the mask covers. In these cases, you must divide by the sum of only those weights that you use from the mask. For the example shown in Figure 4, the output intensity for the pixel `e` is given by:

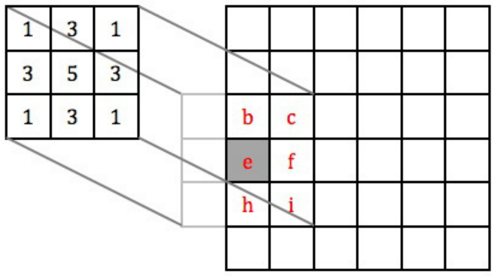


Figure 4: If the mask hangs over the edge of the image, use only those mask values that cover the image in the weighted sum.

```
(3b + c + 5e + 3f + 3h + i)  / 16
```

Figure 5 shows a sample image blurred using the following masks, which are given in the handout as `blur-slightly-mask.txt` and `blur-mask.txt`, respectively:

```
1 3 1           1 2 3 2 1
3 5 3           2 3 4 3 2
1 3 1           3 4 5 4 3
                2 3 4 3 2
                1 2 3 2 1
```

The `mask` passed to the function must have width $n \times n$, where n is given by the argument `maskwidth`. If the supplied image does not match the size given by width and height, or if the mask is not square or does not match the size given by maskwidth, or if the maskwidth is not odd, or if the mask contains negative integers or a zero at the origin, your program should abort with a precondition failure when compiled and run with the `-d` flag.

Task 4 (6 pts) *Create a C0 file* `blur.c0` *implementing a function* `blur`*. You may include any auxiliary functions you need in the same file, but you should not include a* `main()` *function.*

You should look at `README.txt` to see how to compile and run this transformation against `blur-main.c0`. You are also strongly encouraged to write some test cases for your programs in a file `images-test.c0`.

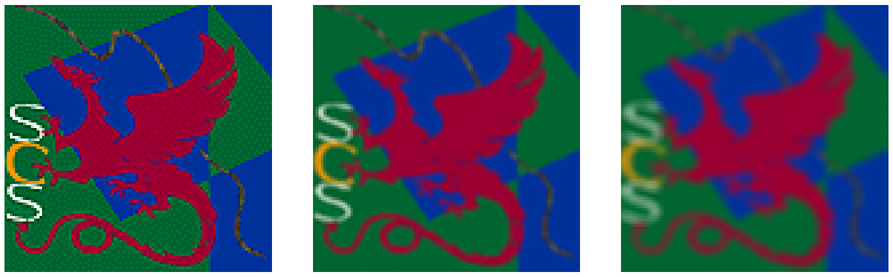


Figure 5: The SCS Dragon: original image (left), blurred slightly with a 3×3 mask (middle), and blurred with a 5×5 mask (right). See text for mask values.

2.4 Your own image processing algorithm (Optional)

In this task, you will perform an image manipulation of your choice. The core of this transformation are three functions:

```
int result_width(int width, int height)
int result_height(int width, int height)
pixel[] manipulate(pixel[] pixels, int width, int height)
```

If `I` is the representation of an image with width `w` and height `h`, then the result of calling `manipulate(I,w,h)` should the representation of image of width `result_width(w,h)` and height `result_height(w,h)`.

Task 5 (Optional) *Create a C0 file* `manipulate.c0` *implementing the three functions described above:* `result_width`, `result_height`, *and* `manipulate`. *You may include any auxiliary functions you need in the same file, but you should not include a* `main()` *function. You may not add arguments to* `manipulate`, *but you can write a separate function* `my_manipulate` *(or whatever) and then call your function from the* `manipulate` *function with some specific arguments.*

You should look at `README.txt` to see how to compile and run this transformation against `manipulate-main.c0`.

If you choose to do this task, be creative! Submissions will be displayed on the Autolab scoreboard and we will make an effort to highlight exemplary submissions. If you include a (small!) file `manipulate.png`, we'll run your transformation against that image; otherwise we'll run your transformation on `g5.png`.

Figure 6: Manipulate me!